AMIGA

GFA BASIC 3.0

Compiler

Alle Rechte vorbehalten. Kein Teil dieses Handbuches darf ohne schriftliche Genehmigung der GFA Systemtechnik GmbH, in welcher Form und mit welchen Mitteln auch immer, sei es auf elektronischem oder mechanischem Weg, kopiert, vervielfältigt oder übermittelt werden. Dies gilt auch für Fotokopien und Tonbandaufnahmen, es sei denn, sie seien für den persönlichen Gebrauch des Käufers bestimmt.

Das auf der beiliegenden Diskette enthaltene Programm GFA-BASIC 3.0 Compiler darf, außer zum persönlichen Gebrauch des Käufers, nicht kopiert werden.

Compilierte GFA-BASIC-Programme dürfen ohne Lizenzgebühren an die GFA Systemtechnik vertrieben werden. Es muß jedoch im Programm, auf der Diskette oder im Handbuch GFA-BASIC 3.0 erwähnt werden.

Der Verlag hat alle Sorgfalt walten lassen, um vollständige und akkurate Informationen zu veröffentlichen. GFA Systemtechnik übernimmt keine Gewähr dafür, daß die beschriebenen Verfahren, Programme usw. funktionsfähig und frei von Schutzrechten Dritter sind.

Copyright © 1989 GFA Systemtechnik GmbH

Programmautor: Frank Ostrowski
Handbuchautor: Uwe Wagner

Änderungen des Textes vorbehalten, Inhalt ohne Gewähr.

1. Auflage, September 1989

Inhaltsverzeichnis

1. Einführung ... 5
1.1 Schnelleinstieg ... 5
1.2 Die Aufgabe eines Compilers ... 6
1.3 Nicht verarbeitbare Anweisungen ... 7
1.4 Arbeiten mit der Shell ... 7
1.4.1 Das "File"-Menü ... 7
1.4.2 Das "Optionen"-Menü ... 8
1.4.3 Das "Sets"-Menü ... 9
1.4.4 Das Shell-Listing ... 11
1.5 Arbeiten im CL1 ... 12

2. Der Compiler ... 15
2.1 Feldindexüberwachung ... 15
2.2 Integerüberlauf ... 16
2.3 Die Compileroptionen ... 19
2.3.1 Übersicht über die Optionen und ihre Einstellung ... 19
2.3.2 Integerdivision (%0; %3) ... 20
2.3.3 Integermultiplikation (*&) ... 22
2.3.4 Speicherplatzreservierung (m...) ... 23
2.3.5 Funktions-Rückgabewert (F%) ... 24
2.3.6 Extern hinzugelinkte Routinen (X) ... 24
2.3.7 Überwachung der Stop-Tasten und EVERY/AFTER (U) ... 25
2.3.8 Interrupt-Routinen (I+; I-) ... 26
2.3.9 SELECT-CASE-Parameter (S&; S%) ... 26
2.3.10 SELECT-CASE-Optimierung (S>; S<) ... 28
2.3.11 Fehlermeldungen (E$; E#) ... 30
2.3.12 Unterroutinen (P>; P<) ... 30
2.3.13 ENDFUNC-Erzeugung (F>; F<) ... 30
2.3.14 Retten der Register A3 - A6 (C+; C-) ... 31
2.3.15 Ein- und Ausschalten zusätzlicher Überlaufprüfung (N+; N-) ... 31
2.3.16 Integeraddition (%6) ... 32

3. Der Linker 35
3.1 Die Linkeroptionen 35
3.1.1 Übersicht über die Linkeroptionen und ihre Einstellung 35
3.1.2 Symboltabelle (-s) 35
3.1.3 Bibliothekauswahl (+ Lib_name, -lLib_name) 36
3.1.4 Linken von Objektfiles 36
3.1.5 TEST.O nicht linken (#) 36
3.1.6 Fehlermeldungen des Linkers 36
3.2 Einbinden von C-Funktionen 37
3.2.1 Einige Besonderheiten 41

4. Programmoptimierung 43
4.1 Einfache Additionen 44
4.2 Multiplikation 46
4.3 Division 49
4.4 Kompliziertere Rechnungen 51
4.5 Schleifen 54
4.6 Zeichenketten 58
4.7 Lokale und globale Variablen 60

1. Einführung

1.1 Schnelleinstieg

In diesem Abschnitt wird beschrieben, wie Sie ein GFA-BASIC 3.0-Listing in ein Programm umwandeln. Dieser Abschnitt ist für eilige Leser gedacht, die den Compiler möglichst rasch ausprobieren möchten.

Bevor Sie dies tun, sollten Sie jedoch eine Sicherheitskopie Ihrer Originaldiskette anfertigen.

Zur Entwicklung eines Programms benötigen Sie folgende Dateien:

GFABASIC	Der GFA-BASIC 3.0-Interpreter
GFA_BCOM	Der GFA-BASIC 3.0-Compiler
GL	Der GFA-BASIC 3.0-Linker
GFALibrary	Die Bibliothek des Linkers
GFALibrary.Index	Das Indexfile der Linkerbibliothek
MENU	Die Shell zur Steuerung des GFA-Entwicklungssystems
***.GFA**	GFA-BASIC 3.0-Sourcefiles, die Sie kompilieren wollen

Um eines der GFA-BASIC-Sourcefiles in ein Programm zu verwandeln, das ohne den Interpreter lauffähig ist, können Sie folgendermaßen vorgehen:

1.) Starten Sie das Programm MENU.

2.) Wählen Sie in der erscheinenden Fileselectbox die zu kompilierende GFA-Datei. Das BASIC-Programm wird nun kompiliert und gelinkt.

3.) Das BASIC-Programm befindet sich jetzt unter dem Namen TEST auf dem Laufwerk. Sie können es starten, indem Sie im Menü "File" den Eintrag "Test" wählen oder das Menüprogramm verlassen und das Programm TEST starten.

4.) Um ein weiteres Programm zu kompilieren, wählen Sie im Menü "File" den Eintrag "Auswahl".

1.2 Die Aufgabe eines Compilers

Ein Computer ist nicht in der Lage, BASIC-Anweisungen direkt zu verstehen. Daher muß ein BASIC-Programm in eine Sprache übersetzt werden, die der Rechner versteht. Ein BASIC-Interpreter liest deshalb Zeile für Zeile eines Programms, übersetzt jede Zeile in Befehle, die für den Rechner verständlich sind und führt diese Befehle dann aus. Das englische Wort "Interpreter" bedeutet in der Übersetzung "Dolmetscher".

Dieses Verfahren hat Vor- und Nachteile. Die beiden wichtigsten Nachteile sind: 1.) Das BASIC-Programm kann nur mit dem Interpreter zusammen laufen. 2.) Dadurch, daß ein BASIC-Befehl zuerst übersetzt und dann erst zur Ausführung gelangt, wird das Programm relativ langsam abgearbeitet.

Ein Compiler ist ein Programm, das sämtliche Befehle eines BASIC-Programms in rechnerverständliche Befehle überführt, ohne die übersetzten Befehle auszuführen. Die übersetzten Befehle werden dann als Programm abgespeichert.

Das englische Wort "compile" bedeutet "zusammenstellen". Ein Compiler setzt also aus den einzelnen BASIC-Anweisungen ein ablauffähiges Programm zusammen. Dieses Programm kann unabhängig vom Interpreter und Compiler arbeiten, und es ist schnell, weil die Befehle schon in übersetzter Form vorliegen.

Der GFA-BASIC 3.0-Compiler, der in diesem Handbuch beschrieben wird, kann ausschließlich GFA-BASIC-Programme der Version 3.0 übersetzen. Diese Programme müssen im GFA-BASIC 3.0-Interpreter mit dem Kommando "SAVE" abgespeichert werden. Der Compiler kann keine mit SAVE,A abgespeicherten Programme verarbeiten.

1.3 Nicht verarbeitbare Anweisungen

Eine Reihe von Anweisungen können vom Compiler nicht umgesetzt werden. Dies sind z.B. diejenigen Anweisungen, die etwas mit dem Auflisten von Programmzeilen zu tun haben, nämlich TRON, TROFF und TRACE$.

Da das kompilierte Programm Variablen nicht mehr unter ihrem Namen kennt, ist auch DUMP nicht möglich.

Das Laden, Speichern und Listen des Programmlistings ist im kompilierten Programm ebenfalls nicht möglich. Die Anweisungen LOAD, SAVE, PSAVE, LIST und LLIST können von einem Kompilat daher auch nicht ausgeführt werden.

1.4 Arbeiten mit der Shell

Ein sogenanntes Shell-Programm erlaubt es, mehrere Programme auf einfache Weise aufzurufen und dabei Parameter an diese Programme zu übergeben. Nach der Beendigung eines Programms, das von der Shell aus aufgerufen wurde, befindet sich der Benutzer wieder in der Shell.

Die Shell zum GFA-BASIC 3.0-Compiler, deren Quelltext auf der Diskette ist, erlaubt den Aufruf des Interpreters, Compilers und Linkers sowie das Einstellen der Compiler- und Linkeroptionen.

1.4.1 Das File-Menü

Im Pulldownmenü mit dem Namen "File" finden Sie als ersten Eintrag "Auswahl". Nach dem Anwählen dieses Eintrags erscheint eine Fileselectbox, in der Sie das GFA-File auswählen können, das Sie bearbeiten möchten. Sie können diesen Eintrag sowie alle anderen auch über Tastatur auswählen. Der Menüpunkt "Auswahl" wird mit Control + A aufgerufen.

Der Menüpunkt "Compiler" übergibt das unter Auswahl festgelegte GFA-File an den Compiler, der daraus ein Objektfile mit Namen TEST.O erzeugt. Der Compiler kann auch mit Control + C aufgerufen werden.

Der nächste Menüpunkt dient dem Aufruf des Interpreters, an den das ausgewählte GFA-File übergeben wird. Diese Funktion kann auch mit Control + I aktiviert werden.

Der Eintrag "Linker", der mit Control + L angesprochen werden kann, linkt die vom Compiler erzeugte Datei TEST.O und erzeugt aus ihr das ablauffähige Programm TEST.PRG. Die Wahl des Eintrags mit dem Namen "Test" startet das erzeugte Programm. Control + T hat denselben Effekt.

Die Taste F10 ruft nacheinander den Compiler und den Linker auf, ist also eine Abkürzung für Control + C und Control + L.

1.4.2 Das "Optionen"-Menü

Im nächsten Pulldownmenü finden Sie die Compiler- und Linkeroptionen. Diese Optionen werden in den nachfolgenden Kapiteln beschrieben. Die folgende Tabelle listet diese Optionen auf. Sie finden in ihr jeweils den dazugehörenden Eintrag und die Tastenkombination in der Shell, die die Option manipuliert.

Eintrag	Tastenaufruf	Option
Interrupts	Rechte Amiga-Taste+I	I+
Select	Rechte Amiga-Taste+S	S& oder S< oder beide
Functions	Rechte Amiga-Taste+F	F<
Procedures	Rechte Amiga-Taste+P	P>
IntDiv	Rechte Amiga-Taste+/	%3
IntMul	Rechte Amiga-Taste+*	*&
Error	Rechte Amiga-Taste+E	E$
Memory	Rechte Amiga-Taste+M	mxxxx
DebugSym	Rechte Amiga-Taste+D	-s

Durch Drücken der entsprechenden Tastenkombination oder Anwählen des Eintrags aktivieren und deaktivieren Sie die Optionen. Bei der Wahl von "Memory" werden Sie nach dem Speicherplatz gefragt, den das Programm belegen soll.

Alle diese Optionen, bis auf DebugSym, betreffen den Compiler. Diejenigen Einträge, hinter denen in der Tabelle mehrere Möglichkeiten aufgeführt wurden, beeinflussen zwei Optionen. Durch einmaligen Aufruf dieser Menüpunkte rufen Sie die erste Option auf, durch zweimaliges Anwählen die zweite und durch dreimaliges Anwählen beide.

Die aktuell eingestellten Optionen werden in der linken oberen Bildschirmecke eingeblendet.

1.4.3 Das "Sets"-Menü

Im letzten Menü mit dem Namen "Sets" können Sie einige Parameter einstellen. Diese Parameter werden im folgenden beschrieben.

Der Parameter WAIT legt fest, ob vor dem Öffnen einer Datei beim Compilieren und Linken auf einen Tastendruck gewartet werden soll oder nicht. Solche zu öffnenden Dateien sind z.B. das zu kompilierende Programm, das Objektfile, GFALibrary.Index, GFALibrary und das Programm-File.

Der Sinn dieser Einstellung besteht darin, daß man dann, wenn auf einen Tastendruck gewartet wird, einen Diskettenwechsel vornehmen kann. Dies ist eventuell notwendig, sofern man nur ein Diskettenlaufwerk hat und nicht alle benötigten Dateien auf einer Diskette untergebracht sind.

Nach dem Anwählen dieses Punktes werden sämtliche Parameter des Menüs "Sets" auf dem Bildschirm angezeigt. An dieser Anzeige erkennen Sie, ob Sie WAIT durch das Anklicken des Menüpunkts ein- oder ausgeschaltet haben.

Auch den nächsten Menüpunkt "MOVE" können Sie durch Anklicken ein- und ausschalten. Wenn Sie MOVE aktiviert haben, wird "MOVE = ON" auf dem Bildschirm angezeigt.

Das Einschalten von MOVE führt dazu, daß der Compiler während des Compilierens sparsamer mit dem verfügbaren Speicher umgeht. Er lädt zu Beginn seiner Tätigkeit den Source-Code des zu bearbeitenden Programms in den Speicher und beginnt die Kompilation, wobei das Kompilat bis zu seiner Abspeicherung ebenfalls im Speicher gehalten wird.

Falls Sie ein sehr großes Programm, aber nur wenig freien Speicher haben, kann es passieren, daß der Source-Code und das Programm gemeinsam zu viel Platz belegen, um komplett im RAM abgelegt zu werden.

Bei aktivem MOVE wird daher der Source-Code jeder übersetzten Prozedur aus dem Speicher entfernt. Dieses "Packen" des Speichers während des Kompilierens kostet natürlich Zeit, daher sollten Sie MOVE normalerweise nicht einsetzen.

Wenn Sie den Eintrag OBJ wählen, können Sie den voreingestellten Namen des Objektfiles verändern, das der Compiler erzeugt (normalerweise ist das TEST.O). Analog dazu können Sie mit PRG den voreingestellten Programmnamen, den der Linker dem fertigen Programm gibt, ändern (normalerweise ist das TEST).

Mit Hilfe von LIB können Sie den Namen der Bibliothek festlegen, die verwendet werden soll (normalerweise GFALibrary). Diese Optionen können Sie natürlich nicht nur für die Benennung der jeweiligen Datei verwenden, sondern z.B. auch für die Festlegung der Pfade benutzen, die zu diesen Dateien gehören sollen.

Die Parametereinstellung können Sie auch über Tastatur aufrufen, indem Sie den entsprechenden Buchstaben eintippen. Also:

```
W oder w für WAIT
M oder m für MOVE
O oder o für OBJ
P oder p für PRG
L oder l für LIB
```

1.4.4 Das Shell-Listing

Das Listing der Shell ist auf der Diskette zu finden. Sie können in diesem Listing die im vorhergehenden Abschnitt beschriebenen Parameter ebenfalls einstellen. Außerdem finden Sie im vorderen Teil dieses Listings die Zeilen

```
gfaint$="GFABASIC"
gfacom$="GFA_BCOM"
gfalnk$="GL"
```

in denen die voreingestellten Namen von Interpreter, Compiler und Linker festgelegt sind. Diese Zeilen können Sie natürlich ändern.

Die Parameter OBJ, PRG und LIB entnehmen ihre Voreinstellung aus den Zeilen

```
tobj$="TEST.O"
tprg$="TEST"
tlib$="GFALibrary"
```

Die Parameter WAIT und MOVE werden im Listing der Shell durch die beiden bool'schen Variablen tmove! und twait! kontrolliert.

Die Zeilen

```
coi&=0        !no I
cos&=3        !S& und s<
cof&=0        !F<
cod&=0        !no %3
com&=0        !no *&
coe&=0        !no E
cop&=0        !no P>
dbsym&=0      !no -s
```

legen die voreingestellten Compiler- und die Linkeroption fest.

1.5 Arbeiten im CLI

Sie können den Compiler und Linker nicht nur mit Hilfe der Shell steuern, sondern z.B. auch im CLI. Die Steuerung im CLI wird in diesem Abschnitt beschrieben.

Die Parameter für den Compiler können Sie in der Form übergeben, in der Sie auch in der Anzeige der Shell erscheinen, z.B.

```
gfa_bcom name S& %3 -m -w
```

Ähnlich ist es bei den Parametern des Linkers. Das Linken mit Erzeugung einer Symboltabelle wird durch

```
gl -s
```

erreicht. Um eine Datei mit dem Namen c_tst mit zu linken, können Sie

```
gl c_tst -l -p
```

angeben. In diesem Fall wird test.o, c_tst.o und GFALibrary gelinkt. Das Symbol # (Doppelkreuz) in der Parameterliste verhindert, daß die Datei test.o mit gelinkt wird. Die Zeile

```
gl # c c_tst
```

würde die Dateien c.o, c_tst.o und gfa3blib linken, nicht aber test.o. Um eine andere Bibliothek als GFALibrary zu verwenden, können Sie deren Namen mit einem vorangestellten Pluszeichen angeben. So führt z.B.

```
gl +neu_lib
```

dazu, daß test.o und neu_lib gelinkt werden. Der Linker betrachtet jeden Parameter, vor dem kein Plus- oder Minuszeichen steht und der nicht -s oder # lautet, als den Namen eines zu linkenden Objektfiles.

Die Reihenfolge, in der die Optionen angegeben werden, ist nicht von Bedeutung, d.h. gl -s tst.o hat denselben Effekt wie gl tst.o -.

2. Der Compiler

2.1 Feldindexüberwachung

Wenn Sie in Ihren Programmen Felder (Arrays) verwenden wollen, so müssen Sie diese mit DIM einrichten. Dabei muß angegeben werden, wie der größte Feldindex lautet. Während des Programmlaufs wird vom GFA-BASIC-Interpreter geprüft, ob Sie einen unzulässig hohen Wert als Feldindex angeben. In einem solchen Fall wird der Fehler 16 (Feldindex zu groß) ausgelöst.

In einem kompilierten Programm wird diese Feldindexprüfung nicht durchgeführt. Falls ein zu großer Wert als Feldindex auftritt, wird keine Fehlermeldung ausgelöst. Die Nicht-Überwachung der Feldindizes führt dazu, daß Programme kürzer und schneller werden.

Das folgende Beispiel soll zeigen, welchen Effekt die fehlende Feldindexüberwachung hat. Das Programm schreibt im Interpreter die Zahlen zwischen 0 und 5 auf den Bildschirm und löst dann die Meldung "Feldindex zu groß" aus. Die Fehlermeldung wird ausgelöst, weil bei der Dimensionierung der maximale Feldindex auf 5 festlegt wurde, in der FOR-Schleife aber der Versuch gemacht wird, das Feldelement x(6) anzusprechen.

```
DIM x(5)
FOR i=0 TO 6
  x(i)=i
  PRINT x(i)
NEXT i
```

Wenn Sie dieses Programm kompilieren und starten, so schreibt es die Zahlen zwischen 0 und 6 auf den Bildschirm, und es erscheint keine Fehlermeldung. Das Programm verhält sich also so, als hätten Sie das Array x() in der ersten Programmzeile ausreichend groß dimensioniert. Sie können an dem Verhalten des kompilierten Programmes nicht erkennen, daß es einen schweren Fehler enthält.

Das Beispiel führt zu der Frage, was das kompilierte Programm tut, wenn ein zu großer Feldindex verwendet wird. Der DIM-Befehl reserviert Speicherplatz für die angegebene Zahl von Arrayelementen. Aus dem Feldindex wird, ohne daß der Programmierer dies zu beachten hat, die Adresse des Feldelements berechnet.

Dies geschieht beim kompilierten Programm auch dann, wenn der Feldindex zu groß ist. Die dabei berechnete Adresse liegt außerhalb des Speicherbereichs, der für das Array reserviert wurde. Bei einem zu großen Feldindex werden deshalb Daten überschrieben, die außerhalb des Bereichs liegen, der für das Feld vorgesehen ist.

In der Regel liegen in dem so überschriebenen Speicherbereich Daten, die für eine fehlerfreie Durchführung des restlichen Programms benötigt werden. Wenn diese Daten überschrieben werden, kann das Programm Fehler produzieren oder sogar abstürzen.

Bis jetzt wurden die Nachteile der fehlenden Indexüberwachung dargestellt. Ein zu großer Feldindex tritt bei einem fertig entwickelten Programm aber nicht auf, da der Programmierer solche Fehler beseitigt hat. Der Interpreter, in dem die Programmentwicklung stattfand, hat ihn bei der Beseitigung solcher Fehler durch die Feldindexüberwachung unterstützt.

Im kompilierten Programm ist eine Indexüberwachung daher überflüssig, denn sie kostet Zeit und Platz. Der Compiler ist bei einfachen Operationen mit Feldelementen (Zuweisungen, Addition zweier Arrayelemente, usw.) zehn- bis elfmal so schnell wie der Interpreter.

2.2 Integerüberlauf

Bei der Veränderung von Integervariablen prüft der GFA-BASIC-Interpreter, ob der neue Wert der Variablen innerhalb des Zahlenbereichs dieses Variablentyps liegt. Die zulässigen Zahlenbereiche der drei Integervariablentypen sind:

Typ	Postfix	Minimum	Maximum
Byte	|	0	255
Word	&	-32768	32767
Long	%	-2147483648	2147483647

Wenn der zulässige Zahlenbereich überschritten wird, so löst dies (je nachdem, welcher Variablentyp vorlag) den Fehler 2, 3 oder 4 aus (Zahl, nicht Integer, Byte oder Wort). Ein solcher Fehler wird Integerüberlauf genannt.

In einem kompilierten Programm wird nicht geprüft, ob ein Integerüberlauf vorliegt. Eine solche Überlaufprüfung ist in einem fehlerfreien Programm überflüssig, sie kostet dort nur Zeit und Platz. Sie ist nur während der Programmentwicklung, die mit dem Interpreter erfolgt, sinnvoll.

Das Programm

```
x&=10000
y&=4*x&
PRINT y&
```

erzeugt im Interpreter die Fehlermeldung "Zahl nicht Wort", während das kompilierte Programm den Wert -25536 auf den Bildschirm schreibt und keine Fehlermeldung ausgibt.

Der Compiler erzeugt aus den ersten beiden Zeilen des Programms folgenden Code (natürlich ohne die Kommentare):

```
move.w #$2710,-$8000(a5)  ;$2710 ist dezimal 10000, -$8000(a5)
                          ;ist die Adresse von x&, der Befehl
                          ;entspricht also der Zeile x&=10000.
move.w -$8000(A5),d0      ;x& wird ins Datenregister 0 geschrieben.
asl.w  #2,d0              ;Die Multiplikation mit einer Zweierpotenz
                          ;(hier 4) wird durch einen schnellen
                          ;Bitschiebebefehl ausgeführt.
move.w d0,-$7ffe(A5)      ;-$7ffe(a5) ist die Adresse von y&. Das
                          ;Ergebnis der Multiplikation wird also
                          ;y& zugewiesen.
```

Der Vorteil der fehlenden Integerüberlaufprüfung besteht in der hohen Geschwindigkeit des erzeugten Programms. Ein Programm mit einer FOR-Schleife, einfachen Additionen und Subtraktionen von Integervariablen ist als kompiliertes Programm etwa um den Faktor 19 schneller als im Interpreter (siehe Tabelle in 5.1).

2.3 Die Compileroptionen

2.3.1 Übersicht über die Compileroptionen und ihre Einstellung

In diesem Abschnitt wird zunächst kurz beschrieben, welche Compileroptionen es gibt und wie Sie diese Optionen beeinflussen können. Anschließend wird jede Option in einem eigenen Abschnitt beschrieben.

%0	Integerdivisionen nur dann als Integerdivision durchführen, wenn der Compiler erkennt, daß das Ergebnis als Integer verwendet wird.
%3	Integerdivisionen immer als Integerdivisionen durchführen.
mxxxx	Das Programm soll nur xxxx Bytes benutzen.
*&	Langwortmultiplikation mit muls durchführen.
*%	Langwortmultiplikation nicht mit muls durchführen.
F%	Rückgabewert einer Funktion soll Integer sein.
X name	Die Routine name aus einer gelinkten Datei entnehmen.
U	Einmal Control+Shift+Alternate, EVERY und AFTER prüfen.
U+	Hinter jedem Befehl C+S+A, EVERY und AFTER testen.
U-	Die Prüfung von C+S+A, EVERY und AFTER testen.
I+	Interrupt-Routinen einschalten.
I-	Interrupt-Routinen ausschalten.
S&	Parameter von SELECT und CASE als Zwei-Byte-Werte behandeln.
S%	Parameter von SELECT und CASE als Vier-Byte-Werte behandeln.
S>	Optimiert SELECT-CASE bezüglich der Ausführungszeit.
S<	Optimiert SELECT-CASE bezüglich der Programmlänge.
E$	Fehlermeldungen als Text.
E#	Fehlermeldungen als Nummer.
P>	Unterroutinen als GFA-BASIC-Unterroutinen kompilieren.
P<	Unterroutinen als 68000er-Unterroutinen kompilieren.
F> F<	ENDFUNC wird erzeugt/wird nicht erzeugt.
C+ C-	Register A3-A6 auf Stapel retten/nicht auf Stapel retten.
N+ N-	Zusätzliche Überlaufprüfung einschalten/ausschalten.
%6	Gemischte Integeraddition als Fließkommaaddition ausführen.

2.3.2 Integerdivision

Die Compileroptionen $%0 und $%3 haben einen Einfluß auf die Division von Integervariablen (Langworte). Wenn die Option %0 aktiv ist, werden die zu dividierenden Variablen in Fließkommavariablen überführt und dann dividiert. Das Ergebnis einer solchen Division ist eine Fließkommavariable.

Falls die Option $%3 eingestellt ist, so werden die beiden Integervariablen nicht in Fließkommavariablen überführt. Das Ergebnis der Integerdivision wird dann als Integerwert interpretiert.

Beispiel: Das Programm

```
$%0
x%=5
y%=2
PRINT x%/y%
```

erzeugt als kompiliertes Programm die Ausgabe 2.5. Die Tatsache, daß eine Nachkommastelle ausgegeben wurde, zeigt Ihnen an, daß x% und y% in Fließkommazahlen überführt wurden, bevor die Division stattfand. Mit der Compileroption $%3 erhalten Sie 2 als Ergebnis, da x% und y% als Integer ohne Nachkommastellen interpretiert wurden und daher auch nur eine Integerdivision durchgeführt wurde. Im Interpreter hat diese Option natürlich keinen Effekt. Sie erhalten in beiden Fällen 2.5.

Der Code, der bei den beiden Optionen erzeugt wird, zeigt den Unterschied noch einmal.

Die Zeilen

```
$%0
a%=x%/y%/z%
```

erzeugen den Code

```
move.l    -$7ffc(a5),d0
bsr       FITOF
move.l    d0,-(a7)
move.w    d2,-(a7)
move.w    d1,-(a7)
move.l    -$7ff8(a5),d0
bsr       FITOF
move.w    (a7)+,d4
move.w    (a7)+,d5
move.l    (a7)+,d3
bsr       FXDIV
move.l    d0,-(a7)
move.w    d2,-(a7)
move.w    d1,-(a7)
move.l    -$7ff4(a5),d0
bsr       FITOF
move.w    (a7)+,d4
move.w    (a7)+,d5
move.l    (a7)+,d3
bsr       FXDIV
bsr       FFTOI
move.l    d0,-$8000(a5)
```

In diesem Listing erkennen Sie den dreimaligen Aufruf des Unterprogramms FITOF (integer to float), mit dem die Integerwerte in Fließkommazahlen umgewandelt werden. Da das Ergebnis an eine Integervariable zurückgegeben wird, ist auch die Routine FFTOI (float to integer) notwendig. Die Division erfolgt mit (FXDIV), also mit einer Routine zur Division mit Fließkommaergebnis.

Dagegen gehört zum Programm

```
$%3
a%=x%/y%/z%
```

das Assemblerlisting

```
move.l      -$7ffc(a5),d0
move.l      -$7ff8(a5),d1
bsr         LDIV
move.l      -$7ff4(a5),d0
bsr         LDIV
move.l      d0,-$8000
```

Dieses kurze Listing ruft lediglich eine relativ einfache Unterroutine zur Division zweier Langworte auf (LDIV) und kann daher erheblich schneller verarbeitet werden als unter der Option $%0. Der Geschwindigkeitsunterschied liegt etwa bei einem Faktor von 10 bis 11.

Für den Fall, daß zwei Integervariablen dividiert werden und das Ergebnis ohne weitere Verrechnung an eine Integervariable zurückgegeben wird, hat die Compileroption $%0 keine Bedeutung. Es wird dann reine Integerarithmetik verwendet.

Falls Zwei-Byte-Variablen dividiert werden, wird statt der Routine LDIV der div-Befehl des Motorola 68000-Prozessors eingesetzt (siehe Abschnitt "Division" im Kapitel "Programmoptimierung").

2.3.3 Integermultiplikation

Die Option $*& beeinflußt die Multiplikation von zwei Integervariablen, von denen mindestens eine der beiden eine Vier-Byte-Variable ist. Normalerweise wird in einem solchen Fall eine Unterroutine zur Multiplikation von zwei Vier-Byte-Variablen aufgerufen. Wenn jedoch die Option $*& eingestellt ist, so wird statt dessen der muls-Befehl des Motorola 68000-Prozessors verwendet.

Dies hat zwei Konsequenzen: 1.) Die Programmausführung wird dadurch beschleunigt. 2.) Der muls-Befehl multipliziert zwei Zwei-Byte-Werte miteinander und hat als Ergebnis einen Vier-Byte-Wert. Falls die zu verrechnenden Vier-Byte-Variablen also Werte enthalten, die den Zahlenbereich einer Zwei-Byte-Variable überschreiten, so wird durch die Verwendung

von nur zwei Bytes ein anderes Ergebnis entstehen als ohne die Option $*&.

Beispiel: Die Zeilen

```
$*&
b%=40000
c%=10
a%=b%*c%
PRINT a%
```

erzeugen im Interpreter das erwartete Ergebnis 400000. Im kompilierten Programm wird aber als Ergebnis -255360 gemeldet, da von b% nur zwei Bytes verwendet wurden, in denen ein so großer Wert wie 40000 nicht dargestellt werden kann.

Der Code, der ohne und mit der Option $*& aus der Zeile a% = b%*c% erzeugt wird, zeigt den Unterschied noch einmal:

Mit *&		**ohne *&**	
move.l	-$7ff8(a5),d0	move.l	-$7ff8(a5),d0
move.l	-$7ffc(a5),d1	move.l	-$7ffc(a5),d1
muls	d1,d0	bsr	LMUL
move.l	d0,$8000(a5)	move.l	d0,-$8000(a5)

Normalerweise ist die Option $*%, die zur Benutzung von LMUL führt, voreingestellt.

2.3.4 Speicherplatzreservierung

Mit Hilfe der Compileroption $mx kann ein Programm erzeugt werden, das nach seinem Start nur x Bytes für sich reserviert. Normalerweise reserviert ein Programm, das mit dem GFA-BASIC 3.0-Compiler erzeugt wurde, den gesamten freien Speicherplatz (minus 16 Kb) für sich.

Mit Hilfe des Befehls RESERVE kann ein Programm zwar angewiesen werden, nur einen Teil des freien Speicherplatzes für sich zu benutzen; dies hat aber einen anderen Effekt als die Compileroption $mx. Ein Programm, das z.B. mit der Anweisung RESERVE 25600 beginnt, reserviert nach seinem Start zunächst den gesamten freien Speicher für sich. Dann erst führt es die erste GFA-BASIC-Anweisung aus, nämlich den RESERVE-Befehl, und gibt einen Teil des reservierten Platzes wieder frei.

Die Compileroption $m25600 führt dagegen dazu, daß nicht zuerst der gesamte freie Platz, sondern nur 25600 Bytes (sofern vorhanden), vom Programm benutzt werden. Diese Compileroption muß in Accessories benutzt werden, da diese nur einen Teil des freien Speichers benutzen dürfen. Ein Beispiel für diese Compileroption finden Sie im Kapitel über Accessories.

2.3.5 Funktions-Rückgabewert

Die Option $F% muß als erste Zeile in einer Funktion angegeben werden, um wirksam zu sein. Sie weist den Compiler an, als Rückgabewert dieser Funktion einen Integerwert zu liefern. Die voreingestellte Rückmeldung ist ein Fließkommawert.

2.3.6 Extern hinzugelinkte Routinen

Die Option $X name muß die erste Zeile in einer Prozedur oder Funktion sein. In diesem Fall wird der Inhalt der Prozedur oder Funktion nicht kompiliert, sondern eine Mitteilung für den Linker hinterlassen, die diesem mitteilt, daß er die Routine "name" aus einer hinzuzulinkenden Objektdatei entnehmen muß.

Im Abschnitt "Einbinden von C-Routinen" wird dargestellt, wie Sie diese Option einsetzen können.

2.3.7 Überwachung der Stop-Tasten und EVERY/AFTER

Während des Programmlaufs besteht im GFA-BASIC-Interpreter normalerweise die Möglichkeit, das Programm durch das gleichzeitige Drücken der Control-, der linken Shift- und der Alternate-Taste zu unterbrechen.

Diese Möglichkeit gibt es voreingestellt im kompilierten Programm nicht. Sie können aber durch die Compileroption $Ux die Unterbrechungsmöglichkeit durch Drücken der drei Tasten einschalten. Damit aber überhaupt eine Abfrage der Stop-Tasten möglich wird, muß die Option $I+ aktiv sein.

Mit Prüfung der Stop-Tasten ist auch eine Überprüfung der EVERY-AFTER-Bedingungen verbunden. Darüber hinaus werden Vorbereitungen für ein RESUME/NEXT getroffen. Dazu sollten Befehle, die Fehler verursachen können, zwischen $u eingeschlossen werden. (Genauer gesagt rettet $u den Programmzähler PC, den Stackpointer SP und den BASIC-Stack-Pointer A3. Es dürfen zwischen dem $u und dem Fehler also keine DIMs, ERASEs oder GOSUBs, FNs ausgeführt werden.)

Der Parameter x kann folgende Werte annehmen, wobei es für x die aufgeführten Möglichkeiten gibt:

$U fügt genau eine Prüfung ein.

$U+ fügt hinter jeder Anweisung, die Code generiert, eine Prüfung ein.

$U- schaltet die Prüfung aus.

Das folgende Programm läßt sich in der ersten Schleife nicht durch einen Druck der drei Stop-Tasten abbrechen, wohl aber (nach dem Druck einer Maustaste) in der zweiten Schleife.

```
$U-
REPEAT
UNTIL MOUSEK
```

```
DO
  $U
LOOP
```

Die DO-LOOP-Schleife läßt sich nicht abbrechen, wenn im Programm nur

```
$U+
DO
LOOP
```

steht, da zwar hinter LOOP, aber nicht hinter DO eine Prüfung eingefügt wird, denn DO ist keine code-erzeugende Anweisung.

2.3.8 Interrupt-Routinen

Mit der Compileroption $I + und $I- können Sie die Interrupt-Routinen für

- EVERY und AFTER
- ON COLLISION
- den Druck der Stop-Tasten (Control-Shift-Alternate)

ein-, bzw. ausschalten. Die Benutzung dieser Optionen führt zu einer Verlängerung des Kompilats.

2.3.9 SELECT-CASE-Parameter

Mit Hilfe der Option $S& können Sie erreichen, daß bei SELECT-CASE-Anweisungen die Ausdrücke, die hinter diesen Befehlen stehen, als Zwei-Byte-Parameter behandelt werden. Die voreingestellte Option $S% führt dazu, daß die hinter SELECT und CASE stehenden Werte als Vier-Byte-Parameter verarbeitet werden.

Betrachtet man den erzeugten Code, so erkennt man die Unterschiede. Das Listing

```
SELECT a%
CASE 1
  case_1:
  INC a%
CASE 2
  case_2:
  INC a%
DEFAULT
  default:
  INC a%
ENDSELECT
endsel:
INC a%
```

enthält Marken, um den symbolisch disassemblierten Code besser durchschauen zu können. Hinter jeder Marke steht ein Befehl (INC a%) als Platzhalter für die einzelnen Befehlsblöcke.

Unter Verwendung der Option $S& entsteht daraus folgender Code:

```
            move.l      -$8000(a5),d0
            bra.s       L1
_CASE_1:    addq.l      #1,-$8000(a5)
            bra.s       _ENDSEL
_CASE_2:    addq.l      #1,-$8000(a5)
            bra.s       _ENDSEL
_DEFAULT:   addq.l      #1,-$8000(a5)
            bra.s       _ENDSEL
L1:         cmpi.w      #$,d0               ;CASE-Auswertung
            beq.s       _CASE_1
            cmpi.w      #$2,d0
            beq.s       _CASE_2
            bra.s       _DEFAULT
_ENDSEL     addq.l      #1,-$8000(a5)
```

Das Label L1 wird dabei nicht erzeugt.

Das Listing legt den Vier-Byte-Wert a% zunächst im Register d0 ab. Dann wird die Routine zur Verzweigung in die einzelnen CASE-Befehlsblöcke aufgerufen. Sie ist im Listing durch den Kommentar "CASE-Auswertung" markiert. Dort werden mit cmpi.w's (compare integer words) die hinter CASE stehenden Werte mit dem Inhalt von d0 nur auf Wortlänge verglichen.

Im Falle einer Übereinstimmung wird der Block hinter dem entsprechenden CASE mit beq.s (branch equal) angesprungen. Dieser Block endet jeweils mit dem Sprung hinter den SELECT-ENDSELECT-Block. Hier wird dieser Sprung mit bra.s _ENDSEL vollzogen. Falls kein hinter CASE stehender Wert paßt, wird der DEFAULT-Block aufgerufen. Ohne die Option S& würden die hier stehenden Vergleiche auf Langwortlänge durchgeführt.

2.3.10 SELECT-CASE-Optimierung

Die Optionen $S> und $S< beeinflussen das Optimierungskriterium für SELECT-CASE-Anweisungen. Bei $S> wird eine Optimierung hinsichtlich der Ausführungszeit vorgenommen, bei $S< hinsichtlich der Programmgröße.

Im letzten Abschnitt haben Sie gesehen, daß am Ende der Programmblöcke hinter CASE und der DEFAULT-Anweisung mit bra.s in die Zeile nach ENDSELECT verzweigt wird. Wenn die Programmblöcke jedoch größer sind als im letzten Abschnitt, in dem sie nur aus einem addq bestanden, muß bra statt bra.s verwendet werden.

Beim bra-Befehl ist der Platz, der für die Sprungdistanzangabe benötigt wird, größer als bei bra.s. Um diesen Platz einzusparen, kann man die Option $$< verwenden. Sie führt dazu, daß der bra-Befehl, der am Ende eines CASE-Programmblocks steht, auf den bra-Befehl des nächsten Programmblocks (vom nächsten CASE oder DEFAULT) gesetzt wird, sofern die Sprungdistanz bis dorthin klein genug ist.

Dadurch kann sich eine Kette von kurzen Branch-Sprüngen ergeben, die schließlich hinter ENDSELECT endet. Diese Kette der Sprünge von bra.s auf bra.s wird vom Compiler in einer späteren Optimierungsstufe eventuell wieder verändert.

Da diese Problematik nicht ganz einfach ist, soll folgendes Listing sie veranschaulichen:

```
            move.l      -$8000(a5),d0
            bra         L1
_CASE_1:    addq.l      #$1,-$8000(a5)
            bra.s       L0
_CASE_2:    addq.l      #$1,-$8000(a5)
L0:         bra         L2
_DEFAULT:   addq.l      #$1,-$8000(a5)
            ...
            Viele Befehle
            ...
L2:         bra.s       _ENDSEL
L1:         cmpi.w      #$1,d0          ;CASE-Auswertung
            beq         _CASE_1
            cmpi.w      #$2,d0
            beq         _CASE_2
            bra         _DEFAULT
_ENDSEL     addq.l      #$1,-$8000(d5)
```

Der entscheidende Punkt ist hier die Zeile bra.s L0. Dieser Branch sollte eigentlich hinter ENDSELECT führen. Die Distanz wäre aber wegen der vielen Befehle im DEFAULT-Programmblock für ein bra.s zu lang. Daher wird auf den bra-Befehl des nächsten CASE-Programmblocks verzweigt.

Diese Konstruktion wird zwar langsamer verarbeitet, benötigt aber etwas weniger Platz als der mit $S > erzeugte Code, der auf solche Sprünge verzichtet.

2.3.11 Fehlermeldungen

Falls in einem kompilierten Programm ein Fehler auftauchen sollte, erscheint eine Fehlermeldung, in der die Art des Fehlers beschrieben wird. Sie können über die Optionen E und $E# festlegen, ob diese Meldung als Text oder als Nummer erscheint.

Wenn Sie die Option E verwenden, erscheint die Fehlermeldung als Text. Da die Fehlertexte ins Kompilat aufgenommen werden müssen, wird dieses dadurch etwas länger.

Die Fehlermeldungs-Option gilt immer für das gesamte Programm. Sie kann nicht, wie manche andere Option, in verschiedenen Teilen des Programms verschieden eingestellt sein.

2.3.12 Unterroutinen

Prozeduren und Funktionen werden bei der Verwendung der voreingestellten Option $P> als GFA-BASIC-Unterroutinen kompiliert. Dies erlaubt Parameterübergaben und problemlose rekursive Aufrufe.

Die Option $P< führt dazu, daß Prozeduren und Funktionen ohne Parameter und ohne lokale Variablen als einfache 68000er-Unterroutinen erzeugt werden. Sie können bei rekursiven Aufrufen problematisch werden, da ein Stapelüberlauf eintreten kann. Dafür ist ihre Abarbeitung schneller. Im Fall von $P< ist auch kein RESUME oder RESUME NEXT möglich.

2.3.13 ENDFUNC-Erzeugung

Eine Funktion wird in GFA-BASIC-Listing immer mit ENDFUNC abgeschlossen. Wenn diese Zeile erreicht wird, erscheint die Fehlermeldung 69 (ENDFUNC ohne RETURN), da eine Funktion vorher mit RETURN verlassen werden muß und nicht bis zum ENDFUNC abgearbeitet werden darf.

Wenn Sie die Option $F> verwenden, so gilt das auch für das kompilierte Programm. Die Option $F< verhindert, daß eine solche Fehlermeldung erscheint. Der Code wird dann einfach hinter der Funktion weiter abgearbeitet, was zu unvorhersehbaren Effekten führen kann.

In der Regel sind Programme, die kompiliert werden, ausgetestete Programme. In einem solchen Fall darf der Fehler 69 nicht auftreten. Durch $F< erreichen Sie dann ein etwas kürzeres Programm.

2.3.14 Retten der Register A3-A6

Routinen, die mit C: oder CALL aufgerufen werden, dürfen nicht die Register A3-A6 verändern. Dies ist bei C-Routinen auf 68000er Computern allgemein üblich. Um auch Assemblerroutinen einbinden zu können, die diese Konvention nicht erfüllen, gibt es die Option $C+ und $C-.

Nach $C+ wird vor jedem CALL, C: ein Befehl eingefügt, um die Register A3-A6 zu retten und danach wieder zu restaurieren. Nach $C- (default) unterbleiben diese Extrabefehle, so daß die Routinen diese Register nicht verändern dürfen.

2.3.15 Ein- und Ausschalten zusätzlicher Überlaufprüfung

FOR-NEXT Schleifen mit ganzzahligen Variablen werden bei Überschreiten des Höchstwertes für den betreffenden Variablentyp nicht abgebrochen. FOR i% = 0 TO 255 ergibt also eine Endlosschleife, da zuerst i% inkrementiert wird (BYTE(255 + 1) = 0!) und somit der Vergleich (< = 255) immer erfüllt bleibt. Dafür gibt es die Kompileroption $N+ und $N-.

Mit $N+ wird eine zusätzliche Überlaufprüfung zum NEXT hinzugefügt. Dies bewirkt allerdings zwei zusätzliche Programmbytes und etwa 1 Mikrosekunde an Laufzeitverlust. Deshalb ist diese Option über $N- wieder ausschaltbar.

2.3.16 Integeraddition

```
x=3.5
x&=10
a&=x+x&
b&=x&+x
```

```
a&=TRUNC(-3.5+10)=TRUNC(6.5)=6
b&=10+TRUNC(-3.5)=10+3=7
```

Bei gemischter Addition von Fließkommavariablen, Integervariablen und Integerergebnis wird häufig, je nach Reihenfolge der Summanden, nur der ganzzahlige Anteil der Fließkommazahl verwendet. Dies kann bei ungleichen Vorzeichen zu Abweichungen vom Interpreterergebnis führen.

```
a&=x+x&
```

ist für den Compiler gleichbedeutend mit

```
a&=ADD(x,x&)
```

Das Interpreterergebnis kann man auch auch durch CFLOAT erzwingen.

```
a&=CLOAT(x+x&)
```

Bei der Entwicklung des Interpreters wurden anfänglich nicht alle Folgen bedacht, daß die Konvertierung von FLießkommawerten nach Integer die Nachkommastellen abschneidet, was z.B. für Zufallszahlen bis zu einer Obergrenze sehr nützlich ist. So ergibt in GFA-BASIC x% = RND*640 eine Zahl zwischen 0 und 639, bei Rundung ergäbe sich eine Zahl zwischen 0 und 640 x% = 639.5 wäre 640).

Andererseits sollte weder eine andere Rundungsmethode für den Compiler eingeführt, noch in jedem Fall die erheblich langsamere Fließkommaadditionsroutine aufgerufen werden.

Für den Fall, daß ein Programm an vielen Stellen durch diesen Fehler betroffen wäre und so erheblich Testaufwand verursachen würde, gibt es die Compileroption $%6.

Wenn der Wert nach $& die 6 erreicht, so wird die entsprechende Optimierung abgeschaltet. Dies bedeutet in der Regel ein erheblich längeres und langsameres Programm. Deshalb verwenden Sie diese Option nur im Notfall, und beschränken Sie sich auf den problematischen Programmteil ($%6...$%0).

3. Der Linker

3.1 Die Linkeroptionen

3.1.1 Übersicht über die Linkeroptionen und ihre Einstellung

In diesem Abschnitt werden alle Linkeroptionen kurz vorgestellt und ihre Einstellung beschrieben. In den nachfolgenden Abschnitten wird jede Option in je einem eigenen Abschnitt erläutert.

-s	Symboltabelle hinzulinken.
+NAME	Verwendet die Bibliothek NAME statt GFALibrary
-LNAME	wie +NAME (wie in dem Shell-Programm L).
NAME	Linkt die Objektdatei NAME.O hinzu.
#	Datei TEST.O nicht linken.
-ONAME	Linkt NAME.O statt TEST.O (wie in dem Shell-Programm O)
-PNAME	Erzeugt das Programm NAME (wie in dem Shell-Programm P).
-W	Schaltet Wait-Flag ein (wie in dem Shell-Programm W).

3.1.2 Symboltabelle

Die Option -s veranlaßt den Linker, eine Symboltabelle anzulegen, damit das Programm unter Verwendung von Symbolen disassembliert und debugged werden kann. Aus dem GFA-BASIC-Listing werden die Namen der Prozeduren und der Funktionen sowie Labelnamen als Symbole eingesetzt.

Dabei werden nur die ersten sieben Zeichen eines Prozedur-, Funktions- oder Labelnamens verwendet, vor die ein Tiefstrich gesetzt wird.

3.1.3 Bibliothekauswahl

Mit der Linkeroption +Lib_Name oder -l Lib_Name kann man festlegen, daß der Linker die Bibliothek mit dem Namen lib_Name verwenden soll. Wenn Sie diese Option nicht verwenden, sucht der Linker nach einer Bibliothek mit dem Namen GFALibrary.

3.1.4 Linken von Objektfiles

Um ein Objektfile zu linken, das im MetaComco-Format vorliegen muß, können Sie den Namen dieses Files als Parameter an den Linker übergeben.

Im Abschnitt "Einbinden von C-Funktionen" wird genauer beschrieben, wie Sie die in den Objektfiles enthaltenen Routinen in Ihren GFA-BASIC-Programmen nutzen können.

3.1.5 TEST.O nicht linken

Durch die Angabe des Zeichens # als Parameter kann der Linker dazu veranlaßt werden, die Datei TEST.O nicht zu linken. Der Linker sucht voreingestellt nach einer Datei dieses Namens, da der Compiler jedes von ihm erzeugte Objektfile so benennt.

3.1.6 Fehlermeldungen des Linkers

Der Linker GL gibt ein paar sehr knappe Fehlermeldungen aus:

```
?xxxxxxx     Unbekanntes Symbol xxxxxxx
+xxxxxxx     Symbol Redefinition
>xxxxxxx     16 Bit-Offset zu groß
```

3.2 Einbinden von C-Funktionen

Dieser Abschnitt beschäftigt sich mit der Frage, wie man Routinen, die man in der Sprache C geschrieben hat, in kompilierte GFA-BASIC 3.0-Programme einbinden kann. Die hier beschriebene Methode der Einbindung verlangt, daß der C-Compiler Objektfiles im MetaComco-Format erzeugen kann.

Die grundsätzliche Methode des Einbindens soll jetzt an einem Beispiel gezeigt werden. Die einzubindende Routine soll die Elemente eines Zwei-Byte-Arrays mit den Zahlen von 0 bis n füllen, wobei die Adresse und die Größe des Arrays übergeben wird. Als Übergabeparameter der Arraygröße wird nur eine Zwei-Byte-Variable verwendet, so daß die Arraygröße auf 32267 Elemente begrenzt ist. Die Funktion, die diese Aufgabe erledigt, arbeitet dabei ohne die Verwendung von Bibliotheksfunktionen.

Das GFA-BASIC-Programm soll so aufgebaut sein, daß es auch im Interpreter lauffähig ist. Im Interpreter kann die C-Routine natürlich nicht durch den Linker eingebunden werden. Die Lauffähigkeit im Interpreter wird deshalb dadurch erreicht, daß eine GFA-BASIC-Routine die Aufgabe der hinzuzulinkenden C-Routine erfüllt.

Das GFA-BASIC-Programm sieht folgendermaßen aus:

```
DIM x&(32000),y&(32000)
'
t1%=TIMER
f_gfa(32000)
t2%=TIMER
PRINT "In GFA-BASIC:"'(t2%-t1%)/200
FOR i&=0 TO 39
  PRINT x&(i&),
NEXT i&
'
t1%=TIMER
f_c(V:y&(0),32000)
t2%=TIMER
```

```
PRINT "In Turbo-C  :"'(t2%-t1%)/200
FOR i&=0 TO 39
  PRINT y&(i&),
NEXT i&
'
PROCEDURE f_gfa(n&)
  FOR i&=0 TO n&
    x&(i&)=i&
  NEXT i&
RETURN
'
PROCEDURE f_c(adr%,n&)
  $X fuellen
  i&=0
  a%=adr%
  WHILE i&<=n&
    WORD{a%}=i&
    INC i&
    ADD a%,2
  WEND
RETURN
```

Es werden zunächst zwei Arrays dimensioniert. Anschließend werden zwei Zeitmessungen durchgeführt. Die erste Zeitmessung ruft die Routine f_gfa auf, die den Elementen des Arrays x&() die Werte von 0 bis 32000 zuweist.

Zur Kontrolle werden dann die ersten 40 Elemente des Arrays auf den Bildschirm ausgegeben.

Die zweite Zeitmessung ruft die Routine f_c auf. Diese Routine soll ebenfalls ein Array mit den Werten von 0 bis 32000 füllen, diese Aufgabe soll jedoch im compilierten und gelinkten Programm von einer C-Routine erledigt werden. Diese Routine erhält zu diesem Zweck die Adresse des Feldes und die Anzahl der zu schreibenden Werte.

In der Prozedur f_c finden Sie in der ersten Zeile die Anweisung, die den Linker zum Hinzubinden der C-Funktion "fuellen" veranlaßt. Hinter der Option $X wird der Name der einzubindenden Funktion angegeben. Diese Zeile muß immer die erste Zeile in der Prozedur sein.

Der Linker ersetzt den Inhalt dieser Prozedur durch die hinter $X angegebene C-Routine. Im Interpreter wird diese Zeile ignoriert und die Befehle in den Zeilen hinter $X c_routine ausgeführt. Im Beispiel sind dies drei Zeilen, die dieselbe Funktion wie die C-Routine erledigen und dabei auch wie die C-Routine arbeiten, indem sie die Wertzuweisung über einen Zeiger durchführen.

Nun zum Aufbau des C-Programms, das die Zuweisung durchführt.

```
void fuellen(n,adr)
int  n  ;
int *adr;
{
   int i; int *a;
   a = adr;
   for( i=0 ; i<=n ; *a++ = i++ );
}
```

In der ersten Zeile finden Sie den Namen der Funktion, der im GFA-BASIC-Listing hinter $X auftaucht. Die Funktion hat keinen Rückgabewert und ist daher als void deklariert.

C holt sich diese Parameter in der umgekehrten Reihenfolge vom Stack, also zuerst den Zwei-Byte-Wert n und danach die Adresse einer Zwei-Byte-Variablen adr. Die folgenden Zeilen führen dann die Zuweisung der Werte an die Feldelemente durch.

Damit ist der Aufbau des GFA-BASIC-Listings und des C-Listings bekannt. Nun muß noch geklärt werden, wie Sie den Linker veranlassen können, die C-Funktion an der Stelle $X fuellen einzusetzen. Zu diesem Zweck müssen Sie ihm lediglich den Namen des Objektfiles angeben, das Sie durch den C-Compiler erzeugen ließen.

Diesen können Sie, falls Sie mit dem CLI arbeiten, einfach in die Kommandozeile eintragen. In der mitgelieferten Shell können Sie die Variable OBJ ergänzen.

Es müssen somit folgende Schritte durchgeführt werden:

1.) Im GFA-BASIC-Programm erstellt man eine Prozedur, in der man in der ersten Zeile die Option $X, gefolgt von einem C-Funktionsnamen einsetzt.

2.) Man erstellt ein C-Programm, das diese Funktion enthält. Dabei muß man die Parameter auf dem Stack entgegennehmen lassen. Die Parameter liegen dort, im Vergleich zur Übergabe durch das GFA-BASIC-Programm, in der umgekehrten Reihenfolge.

3.) Beim Linken muß man das Objektfile des C-Listings mit der zu linkenden Routine angeben.

In unserem Beispiel hatte die C-Funktion keinen Rückgabewert und war von daher als void deklariert. Man kann eine C-Funktion aber natürlich nicht nur anstelle einer GFA-BASIC-Prozedur, sondern auch anstelle einer GFA-BASIC-Funktion einsetzen.

Ein entsprechendes GFA-BASIC-Listing wäre z.B.:

```
wert%=100
PRINT @dop(wert%)
'
FUNCTION dop(a%)
  $X verdoppele
  ALERT 1,"Platzhalter für|eine C-Funktion.",1,"Return",a&
  RETURN 0
ENDFUNC
```

In diesem Beispiel übernimmt die GFA-BASIC-Funktion nicht die Aufgabe der hinzuzulinkenden C-Routine. Das simple C-Programm lautet:

```
long verdoppele(x)
long x;
{
   return x+x;
}
```

Der Rückgabewert einer solchen externen Funktion ist immer ein Integerwert.

Sie können natürlich auch Assemblerroutinen linken. Diese müssen ihre Parameter vom Stack übernehmen und dürfen den Stackpointer SP und die Register A3, A4, A5 und A6 nicht verändern.

Wird die oben verwendete C-Routine direkt in Assembler geschrieben, so resultiert das folgende Listing:

```
c_fuellen:  move.w    4(sp),d0
            movea.l   6(sp),a0
            moveq.l   #0,d1
1$:         move.w    d1,(a0)+
            addq.w    #1,d1
            cmp.w     d0,d1
            bls.s     1$
            rts
```

Der Aufruf des Objekt-Files, welches der Assembler liefert, erfolgt genau so wie beim C-Compiler.

3.2.1 Einige Besonderheiten

In größeren Programmen ist es in der Regel erforderlich, keine PC-relativen Aufrufe zu verwenden. Sie sollten in Ihrem C-Compiler die entsprechende Option einstellen.

Die C-Routinen dürfen die Register A3 bis A6 nicht verändern oder bestimmte Werte in ihnen erwarten. Gegebenenfalls brauchen Sie ein Assembler-Interface, wie etwa:

```
name:     movem.l A3-A6,resave
          jsr c_name
          movem.l regsave,A3-A6
          rts
regsave:  ds.l 4
```

Sie müssen dann in Ihrer C-Routine unter Umständen einen Extra-Dummy Parameter verwenden, der die Rücksprungadresse des jsr c_name verdeckt, d.h. statt

```
name (long x,long y)
c_name(long dummy y,long x,long y)
```

4. Programmoptimierung

Der 3.0-Compiler erzeugt Assemblercode. Sie können ihn bei dieser Arbeit durch geeignete Programmierung unterstützen. In diesem Kapitel geht es um die Frage, wie ein GFA-BASIC-Programm aussehen sollte, damit das daraus entstehende kompilierte Programm besonders schnell und kurz wird. Um dieses Ziel zu erreichen, müssen Sie wissen, wie der Compiler bei seiner Optimierung vorgeht.

Zu diesem Zweck werden in diesem Kapitel einige Assemblerlistings aufgeführt, die den vom Compiler erzeugten Code darstellen. Wenn Sie die vom Compiler erzeugten Programme selbst disassemblieren möchten, sollten Sie sich entweder mit Hilfe eines Monitors einen freien Speicherbereich aussuchen (z.B. ab Hex $50000), in den Sie das Programm direkt laden, oder Sie laden das kompilierte Programm als Segment, wobei die meisten Monitor-Programme den Segment-Pointer, d.h. die Startadresse des Segmentes, angeben. Das disassemblierte Programm beginnt immer mit der Assembler-Anweisung

```
jsr INIT (Jump to SubRoutine)
```

daran schließt sich direkt der Programm-Code an. Das Ende dieses Codes wird durch das Label BaseA4 gekennzeichnet, welches im Hex-Dump ersichtlich wird.

Sie werden von diesem Kapitel auch ohne Assemblerkenntnisse profitieren können, indem Sie die aus den Listings abgeleiteten Vorschläge beachten. Sie werden dann jedoch nicht verstehen, warum eine bestimmte Formulierung eines Programms effektiver ist als eine andere.

Das notwendige Assemblergrundwissen zum Verständnis der disassemblierten Listings ist bewußt klein gehalten worden. Eine Zielgruppe dieses Kapitels sind GFA-BASIC-Programmierer, die sich mit Assembler vertraut machen wollen, um z.B. zeitkritische Unterroutinen in dieser Sprache zu schreiben.

Wir können natürlich nicht auf alle möglichen Anweisungen eingehen, die der Compiler verarbeiten kann, sondern wir greifen lediglich einige Beispiele auf, die vom Optimierungsstandpunkt her besonders interessant sind.

Die Beispiele in den folgenden Kapiteln enthalten teilweise Zeitangaben. Dazu ist zu sagen, daß die Dauer, die für die Verarbeitung eines Programms erforderlich ist, im wesentlichen davon abhängt, ob noch andere Tasks laufen oder nicht. Das bedeutet, daß die absoluten Zeiten bei den einzelnen Beispielen nicht so sehr von Bedeutung sind. Sie dienen vielmehr der Verdeutlichung von Zeitunterschieden.

4.1 Einfache Additionen

Der Prozessor des AMIGA, der Motorola 68000, kennt Befehle zur Verarbeitung von Integerzahlen. BASIC-Zeilen, in denen nur Integervariablen miteinander verrechnet werden, können in wenige Assembleranweisungen übersetzt werden, sofern diese Zeilen nicht zu komplex sind.

Für die Verarbeitung von Fließkommavariablen müssen weitaus mehr Assembleranweisungen abgearbeitet werden, was sich in der Rechenzeit niederschlägt. Die Addition zweier Integerzahlen erfolgt etwa drei- bis viermal so schnell wie die Addition zweier Fließkommazahlen.

Die erste Frage, die hier besprochen werden soll, lautet: Wann kann der Compiler die schnellere Integerarithmetik verwenden und wann greift er auf Fließkommaarithmetik zurück?

Im Interpreter werden die Anweisungen a%=b%+c% und a%=ADD(b%,c%) unterschiedlich schnell verarbeitet. Der Compiler produziert in diesem einfachen Fall aus beiden Anweisungen denselben Code, nämlich

```
move.l      -$7ff8(a5),d0
add.l       -$7ffc(a5),d0
move.l      d0,-$8000(a5)
```

Die Adressen -$xxxx(a5) hängen davon ab, wann welche Variablen im Programm verwendet werden. Die Variablen liegen ab -$8000(a5).

Wenn Sie Zwei-Byte-Variablen verwendet haben, z.B. a& = b& + c&, so erzeugt der Compiler die Befehle

```
move.w    -$7ffc(a5),d0
add.w     -$7ffe(a5),d0
move.w    d0,-$8000(a5)
```

Bei Ein-Byte-Variablen (a| = b| + c|) entsteht

```
moveq.l   #$0,d0
move.b    -$7ffe(a5),d0
move.b    -$7fff(a5),d1
add.l     d1,d0
move.b    d0,-$8000(a5)
```

Eine Programmzeile wie z.B. a = b + c verwendet jedoch Fließkommavariablen. Um eine Fließkommaaddition durchführen zu können, enthält das kompilierte Programm eine entsprechende Unterroutine. Im disassemblierten Listing eines Programms mit Symboltabelle finden Sie

```
lea.l     -$7ff0(a5),a0
lea.l     -$7ff8(a5),a1
bsr       VVFADD
lea.l     -$8000(a5),a0
move.l    d0,(a0)+
move.w    d1,(a0)+
move.w    d2,(a0)+
```

In diesem Listing finden Sie die Zeile bsr VVFADD. Diese Zeile verzweigt zu der Fließkommaadditions-Routine, deren Abarbeitung wesentlich länger dauert als die der wenigen Befehle, die z.B. durch a% = b% + c% erzeugt werden.

Der Code, der aus der Zwei-Byte-Addition produziert wird, kann vom Rechner am schnellsten verarbeitet werden. Es folgt der Code der Ein-Byte-Addition, der der Vier-Byte-Addition und schließlich, mit großem Abstand, der Code der Fließkommaaddition.

In der folgenden Tabelle finden Sie ein Beispiel für die Zeitunterschiede der verschiedenen Variablentypen bei einer einfachen Addition im Interpreter und im Compiler. Es wurde jeweils eine Schleife mit 50000 Durchläufen verwendet, wobei die Zeit für die Schleifenbefehle herausgerechnet wurde.

Typ	Interpreter	Compiler	Anweisung
Byte	8,42	0,40	a|=b|+c|
Word	8,64	0,66	a&=b&+c&
Long	7,94	0,40	a%=b%+c%
Float	6,23	1,50	a = b+c

Für die Subtraktion gelten diese Ausführungen in ähnlicher Weise.

4.2 Multiplikation

Die Optimierung, die der Compiler für die Multiplikation verwendet, beinhaltet eine Besonderheit. Der Prozessor des AMIGA kennt zwar Multiplikationsbefehle, aber er benötigt für ihre Ausführung sehr viele Taktzyklen und er kennt diese Anweisungen nur für Zwei-Byte-Werte. Der GFA-BASIC 3.0-Compiler muß daher Multiplikationen einer Konstanten mit einer Vier-Byte-Variablen unter Vermeidung der Assemblerbefehle mulu und muls kodieren.

Die Zeile

```
x%=y%*4
```

kann z.B. besonders einfach in den Code

```
move.l  -$8000(a5),d0
lsl.l   #$2,d0
move.l  d0,-$7ffc(a5)
```

übersetzt werden, da 4 eine Zweierpotenz ist. Die Kodierung der Anweisung

```
x%=y%*21
```

ist schon etwas aufwendiger, da 21 binär gleich 10101 ist. Um den vom Compiler erzeugten Code besser verstehen zu können, nehmen wir in den Kommentaren zum Code einmal an, daß y% = 100 ist. Der erzeugte Code lautet:

```
move.l  -$8000(a5),d0     ;y%, also 100, ins Register d0
move.l  d0,d2             ;eine Kopie von y% ins Register d2
lsl.l   #$2,d0            ;d0 mal 4, d0 enthält nun 400, d2=100
add.l   d0,d2             ;addiere 400 auf 100, also d0=400, d2=500
lsl.l   #$2,d0            ;d0 mal 4, also d0=1600, d2=500
add.l   d2,d0             ;addiere d2 auf d0: d0=2100, d2=500
move.l  d0,-$7ffc(a5)     ;schreibt das Ergebnis 2100 in x%
```

Dies verbraucht zwar 6 Bytes mehr Speicherplatz als ein muls #$15,d0 (das es aber nicht für Vier-Byte-Werte gibt), ist aber mit 52 Taktzyklen Ausführungszeit (ohne den ersten und letzten move-Befehl) schneller.

Die Anweisung

```
x&=y&*21
```

würde dagegen unter Verwendung des muls-Befehls kodiert. Für 100.000 Multiplikationen mit 21 ist die Anweisung mit Vier-Byte-Variablen also etwas schneller:

Anweisung	Interpreter	Compiler
x%=y%*21	14,28	1,20
x&=y&*21	15,26	1,48

Wenn jedoch nicht mit 21 multipliziert wird, sondern mit Werten, deren Zerlegung in die Assembleranweisungen add, sub, lsl und asl einen relativ langen Code erfordern, so wird die Vier-Byte-Variante nicht nur erheblich länger, sondern auch langsamer als die Zwei-Byte-Anweisung, die mit muls arbeitet. Benutzt man statt 21 z.B. 21845 (binär 101010101010101), so lauten die Ausführungszeiten für 100.000 Wiederholungen

Anweisung	Interpreter	Compiler
x%=y%*21845	14,32	2,70
x&=y&*21845	15,38	1,86

Bisher wurde die Multiplikation einer Variable mit einer Konstanten betrachtet. Bei der Multiplikation zweier Integervariablen ist es aber auch von zentraler Bedeutung, daß der mul-Befehl des Prozessors nur Zwei-Byte-Werte verrechnet.

Die folgende Liste zeigt den von verschiedenen Integermultiplikationen erzeugten Code und dessen Ausführungszeit für 250.000 Multiplikationen.

```
move.l  -$7ff4(a5),d0      ;x%=y%*z%, 7,34
move.l  -$7ff8(a5),d1
bsr     LMUL
move.l  d0,-$7ffc(a5)
move.w  -$7ff0(a5),d0      ;x%=y%*z&, 7,48
ext.l   d0
move.l  -$7ff8(a5),d1
bsr     LMUL
move.l  d0,-$7ffc(a5)
move.w  -$7ff0(a5),d0      ;x%=y&*z&, 2,92
muls    -$7fee(a5),d0
move.l  d0,-$7ffc(a5)
move.w  -$7ff0(a5),d0      ;x&=y&*z&, 2.72
muls    -$7fee(a5),d0
move.w  d0,-$7fec(a5)
```

Der große Zeitunterschied zwischen den ersten und den lezten beiden beruht darauf, daß in den letzten Fällen muls statt der Routine LMUL ver-

wendet werden konnte. Die absoluten Zeiten hängen allerdings auch von den Werten ab, die in y%, z%, y& und z& stehen. Die Zeitrelationen zwischen den verschiedenen Anweisungen bleiben aber in etwa gleich.

Man kann in diesen beiden Fällen, in denen eine Vier-Byte-Variable beteiligt ist, mit der Compileroption *& ebenfalls die Benutzung von muls erzwingen. Dabei muß jedoch beachtet werden, daß von den verrechneten Vier-Byte-Werten nur zwei Bytes in die Rechnung eingehen. Eine genauere Beschreibung finden Sie im Kapitel über Compileroptionen (Integermultiplikation).

Die geringen Unterschiede zwischen den beiden LMUL- und den beiden muls-Fällen beruhen nur auf move.w versus move.l und dem ext.l-Befehl. Bei der Multiplikation von Integervariablen ist es also lohnend, Zwei-Byte-Variablen statt Vier-Byte-Variablen zu verwenden.

4.3 Division

Divisionen von Integervariablen können Nachkommastellen aufweisen, z.B. x% = 5, y% = 2, x%/y% = 2.5. Im Unterschied zur Multiplikation müßte der Compiler die Division daher generell als Fließkommadivision durchführen. Dies tut er auch, sofern der Programmierer ihn nicht ausdrücklich dazu auffordert, die Division zweier Integervariablen als Integerdivision mit Integerergebnis durchzuführen.

Diese Aufforderung erfolgt durch die Compileroption $%3. Die komplementäre Option $%0, die dazu führt, daß jede Division als Fließkommadivision durchgeführt wird, ist voreingestellt. Die Analyse des Codes, der durch die Zeile

```
x=y%/z%
```

erzeugt wird, zeigt den Unterschied. Unter der voreingestellten Option %0 entsteht:

```
move.l  -$7ff8(a5),d0
```

```
bsr     FITOF
move.l  d0,-(a7)
move.w  d2,-(a7)
move.w  d1,-(a7)
move.l  -$7ff4(a5),d0
bsr     FITOF
move.w  (a7)+,d4
move.w  (a7)+,d5
move.l  (a7)+,d3
bsr     FXDIV
lea.l   -$8000(a5),a0
move.l  d0,(a0)+
move.w  d1,(a0)+
move.w  d2,(a0)+
```

Sie erkennen, daß die beiden zu dividierenden Variablen mit der Routine FITOF (integer to float) in Fließkommavariablen transformiert werden. Dann wird die Routine zur Division zweier Fließkommawerte aufgerufen (FXDIV) und das Ergebnis der Variable x ab -$8000(a5) zugewiesen.

Wenn dieselbe Zeile unter der Option $%3 kompiliert wird, entsteht folgender Code:

```
move.l  -$7ffc(a5),d0
move.l  -$7ff8(a5),d1
bsr     LDIV
lea.l   -$8000(a5),a0
bsr     ISTOF
```

Die Routine ISTOF erledigt die Zuweisung an x. Hier wird die Routine LDIV zur Division zweier Long-Integer-Variablen aufgerufen. Wenn die zu dividierenden Werte so geartet sind, daß Nachkommastellen anfallen, so werden diese von LDIV natürlich nicht zurückgemeldet.

Lautet die Zeile x% = y%/z%, statt wie hier x = y%/z%, wäre dies ohne Belang, da das Ergebnis an eine Integervariable zurückgemeldet wird. In

einem solchen Fall verwendet der GFA-BASIC-Compiler automatisch die Division mit Hilfe von LDIV, auch wenn $%0 aktiv ist.

Falls Zwei-Byte-Variablen dividiert werden, kann der div-Befehl des Motorola-68000 eingesetzt werden, der natürlich noch schneller als die Routine LDIV arbeiten kann. Der Code ist dann besonders einfach und schnell. Aus der Zeile

```
x&=y&/z&
```

entsteht:

```
move.w  -$7ffe(a5),d0
ext.l   d0
divs    -$7ffc(a5),d0
move.w  d0,-$8000(a5)
```

Daraus folgt: Verwenden Sie nach Möglichkeit bei der Division Zwei-Byte-Variablen und schalten Sie die $%3-Option ein, wenn Sie sicher sind, daß keine Integervariablen dividiert werden, bei deren Verrechnung relevante Nachkommastellen entstehen.

Bei der Division durch Konstanten werden ähnliche Optimierungen vorgenommen wie bei der Multiplikation, so wird z.B. die Division durch Zweierpotenzen in Bit-Schiebe-Anweisungen überführt.

4.4 Kompliziertere Rechnungen

Bisher haben wir nur den Fall betrachtet, in dem zwei Variablen miteinander verrechnet wurden. Nun soll es um den Fall gehen, in dem mehr als zwei Variablen miteinander verknüpft werden. Der Compiler kann solche Befehlszeilen nur bis zu einem bestimmten Komplexitätsgrad in Code umsetzen, der keine Verzweigungen zu Unterroutinen enthält. Das folgende Beispiel soll dies illustrieren.

Um die Programmzeile

```
x&=x&+(y&*y&-y&)/y&+z&
```

fünfzigtausendmal abzuarbeiten, benötigt ein kompiliertes Programm (ohne die Schleifenbefehle) etwa 2.56 Sekunden. Diese Zeile kann aber auch zerlegt werden in

```
x&=x&+(y&*y&-y&)/y&
x&=x&+z&
```

Diese beide Zeilen leisten, vom Ergebnis her gesehen, dasselbe wie die zuerst angegebene Anweisung. Für das 50.000-fache Durchführen dieser Zeilen braucht das compilierte Programm aber nur etwa 1.875 Sekunden. Die Regel, die sich daraus ableiten läßt, lautet: Zerlegen Sie Zeilen mit komplizierteren Integer-Rechnungen in mehrere Zeilen.

Um zu verstehen, warum die zweizeilige Variante schneller verarbeitet wird als die einzeilige, muß man den entsprechenden Ausschnitt des (symbolisch) disassemblierten Programms analysieren. Die Zeile

```
x&=x&+(y&*y&-y&)/y&+z&
```

wird überführt in

```
move.w  -$7ffe(a5),d0
muls    -$7ffe(a5),d0       ;y&*y&
movea.w -$7ffe(a5),a0
sub.l   a0,d0               ;-y&
move.w  -$7ffe(a5),d1
ext.l   d1
bsr     LDIV                ;/y&
movea.w -$8000(a5),a0
add.l   a0,d0               ;+x&
add.w   -$7ffc(a5),d0       ;+z&
move.w  d0,-$8000(a5)
```

Die Adressen der Variablen beruhen wiederum auf der Reihenfolge der Einführung ins Programm. Hier wurde z.B. die Variable x& als erste, y& als zweite und z& als dritte ins Programm eingeführt. Ihre Adressen sind daher

```
x&  -$8000(a5)
y&  -$7ffe(a5)
z&  -$7ffc(a5)
```

Im Listing finden Sie die Zeile bsr LDIV, also den Aufruf einer Unterroutine zur Division zweier Integerwerte. Wenn Sie dieses Listing nun mit dem der zweizeiligen Version vergleichen, so finden Sie keinen Unterroutinenaufruf. Die Anweisungen

```
x&=x&+(y&*y&-y&)/y&
x&=x&+z&
```

führen zu dem Code

```
move.w  -$7ffe(a5),d0
muls    -$7ffe(a5),d0       ;y&*y&
movea.w -$7ffe(a5),a0
sub.l   a0,d0               ;-y&
move.w  -$7ffe(a5),d1
divs    d1,d0               ;/y&
add.w   d0,-$8000(a5)       ;Ende der ersten Zeile
move.w  -$7ffc(a5),d0
add.w   d0,-$8000(a5)       ;+z&
```

Wie Sie sehen, ist dieses Listing ein paar Befehle kürzer als das vorhergehende und enthält keine Unterroutinenaufrufe mit bsr (branch subroutine).

Im Interpreter ist die einzeilige Befehlsvariante schneller als die zweizeilige. Wenn Sie Ihre Programme aber auf den Compiler zuschneiden wollen, dann sollten Sie ihm nur relativ kurze Anweisungen mit Integer-Rechnungen präsentieren, die er optimal in Assemblerbefehle kodieren kann.

Häufig setzt man aus Gründen der Übersichtlichkeit Klammern in Rechnungen ein, die eigentlich überflüssig sind. So würde man z.B. statt

```
x&=x&+(y&*y&-y&)/y&
```

die Zeile

```
x&=x&+((y&*y&)-y&)/y&
```

verwenden. Ob das für dieses Beispiel nun übersichtlicher ist oder nicht, soll uns jetzt nicht beschäftigen. Im Interpreter werden Sie feststellen, daß die Version mit einem zusätzlichen Klammerpaar langsamer ist. Im kompilierten Programm sind beide Versionen aber gleich schnell, da aus ihnen derselbe Code erzeugt wird.

Wenn Sie also aus Gründen der Übersichtlichkeit Klammern setzen wollen, die keinen Einfluß auf das Ergebnis haben, so können Sie das ohne einen Verlust an Geschwindigkeit des kompilierten Programms tun.

4.5 Schleifen

Eine Information zur Programmoptimierung zum 3.0 Interpreter war, daß die FOR-NEXT-Schleife die schnellste war. Beim Compiler zum GFA-BASIC 3.0 ist das anders.

Wenn man eine Leerschleife mit 50000 Durchläufen in den drei Schleifentypen FOR-NEXT, REPEAT-UNTIL und WHILE-WEND durchlaufen läßt, ergeben sich für eine Vier-Byte-Laufvariable folgende Zeiten:

	FOR-NEXT	REPEAT-UNTIL	WHILE-WEND
Interpreter	1,46	7,56	8,30
Compiler	0,40	0,40	0,40

Die DO-LOOP-Schleife mit EXIT IF und die Abwandlungen der DO-LOOP-Schleife, z.B. DO-LOOP UNTIL, sollen hier nicht betrachtet wer-

den. Es soll nur bemerkt werden, daß die DO-LOOP-Schleife mit EXIT IF im Compiler langsamer ist (0.875).

Anmerkung: Wenn man alle Schleifen in einem Programm nacheinander ausführt, wird die zweite Schleife etwa doppelt so lange dauern, wenn zur Angabe der Zeiten jeweils ein Print-Befehl verwendet wird und noch kein Fenster offen ist, da das Öffnen des Fensters teilweise parallel zur Berechnung in der zweiten Schleife erfolgt.

Wie man an der Tabelle sehen kann, ist die FOR-NEXT und die WHILE-WEND-Schleife im compilierten Programm gleich schnell. Es ist im 3.0-Compiler also nicht sinnvoll, ein Programm an den Compiler "anzupassen".

Die Gleichheit der Ausführungszeiten legt natürlich den Verdacht nahe, daß der Code, der durch die drei Schleifen erzeugt wird, sehr ähnlich sein muß. Dieser Verdacht ist richtig, wie die disassemblierten Listings der drei Schleifentypen nun zeigen sollen. Wir haben dabei in jede Schleife noch die Anweisung b% = 10 eingesetzt, damit die Position des Schleifeninhalts im Code erkennbar ist.

Die FOR-Schleife

```
FOR i%=1 TO 10000
  b%=10
NEXT i%
```

erzeugt folgendes Listing (die angesprungenen Adressen und die Variablenadressen dienen wiederum nur als Beispiel):

```
        moveq.l #$1,d0              ;Schleifenstart bei 1.
        move.l  d0,-$7ff8(a5)       ;Startwert i% zuweisen.
L1:     moveq.l #$a,d0              ;Den Wert 10 der
        move.l  d0,-$8000(a5)       ;Variable b% zuweisen.
        addq.l  #$1,-$7ff8(a5)      ;Erhöhe i% um 1.
        cmpi.l  #$2710,-$7ff8(a5)   ;Schleifenendwert erreicht?
        ble.s   L1                  ;Falls noch nicht, dann zurück.
```

Die ersten beiden Anweisungen setzen den Schleifenstartwert i%. Danach folgt der Schleifeninhalt, hier ab dem Label L1, in dem mit zwei Befehlen der Wert 10 der Variable b% zugewiesen wird.

Anschließend wird die Laufvariable i% um 1 erhöht und anschließend geprüft, ob der Schleifenendwert schon erreicht ist. Der Endwert war 10000, also hexadezimal $2710. Falls die Laufvariable noch kleiner oder gleich ist, muß die Schleife noch einmal durchlaufen werden.

Eine Schleife, deren Endwert nicht durch eine Konstante, sondern z.B. durch eine Variable festgelegt wird, hat natürlich einen anderen Aufbau, ebenso wie eine Schleife, die STEP verwendet.

Da nach dem Schleifeninhalt das addq kommt und im Anschluß daran erst die Abbruchbedingung geprüft wird, ist i% nach Durchlaufen der Schleife gleich 10001. Es ist in der Sprache BASIC eine Konvention, daß die Laufvariable einer FOR-NEXT-Schleife nach Verlassen der Schleife größer ist als der Endwert der Schleife. Dies liegt daran, daß das Abbruchkriterium der Schleife erfüllt ist, wenn der Zähler größer als der Endwert ist.

Man sollte sich aber, wenn man seine Programme in andere BASIC-Dialekte oder in andere Sprachen übersetzen will, nicht darauf verlassen, weil diese BASIC-Konvention nicht überall eingehalten wird.

Nun zur REPEAT-UNTIL-Schleife. Aus den BASIC-Zeilen

```
i%=0
REPEAT
  INC i%
  b%=10
UNTIL i%=10000
```

werden folgende Assembleranweisungen:

```
        clr.l   -$7ff8(a5)          ;i%=0.
L1:     addq.l  #$1,-$7ff8(a5)      ;Erhöhe i% um 1.
        moveq.l #$a,d0              ;b% auf den
        move.l  d0,-$8000(a5)       ;Wert 10 setzen.
        cmpi.l  #$2710,-$7ff8(a5)   ;Vergleiche 10000 mit i%.
        bne.s   L1                  ;Falls i% ungleich 10000.
```

In diesem Listing wird der Befehl clr.l nur einmal durchlaufen, alle anderen Befehle viele Male. Im Listing zur FOR-NEXT-Schleife werden dieselben Befehle vielfach durchlaufen wie in der REPEAT-UNTIL-Schleife: ein addq, ein moveq, ein move, ein cmpi und ein branch-Befehl mit logischer Bedingung.

Die gleichen Ausführungszeiten der FOR-NEXT- und der REPEAT-UNTIL-Schleife werden dadurch verständlich. Bei der WHILE-WEND-Schleife sind es ebenfalls die gleichen Befehle, die vielfach abgearbeitet werden. Aus den Zeilen

```
i%=0
WHILE i%<10000
  INC i%
  b%=10
WEND
```

entsteht der Assemblertext

```
        clr.l   -$7ff8(a5)          ;i%=0.
        bra.s   L2                  ;Springe zum cmpi.
L1:     addq.l  #$1,-$7ff8(a5)      ;INC i%.
        moveq.l #$a,d0              ;b% auf den
        move.l  d0,-$8000(a5)       ;Wert 10 setzen.
L2:     cmpi.l  #$2710,-$7ff8(a5)   ;Vergleiche 10000 und i%.
        blt.s   L1                  ;Wenn i% kleiner ist, zurück.
```

Die Umsetzung der WHILE-WEND-Schleife in Assembler unterscheidet sich von der REPEAT-UNTIL-Schleife durch zwei Dinge.

1. Die logische Bedingung ist anders formuliert, da es bei der REPEAT-UNTIL-Schleife eine Abbruchbedingung und bei der WHILE-END-Schleife eine "Fortsetzungsbedingung" ist.
2. Die WHILE-WEND-Schleife ist eine abweisende Schleife. Ihre Bedingung wird also vor dem ersten Durchlaufen des Schleifeninneren getestet. Daher wird im Assemblerprogramm zunächst mit einem bra-Befehl zur Prüfung der Schleifenbedingung gesprungen.

In allen anderen Punkten ist die Realisierung dieser beiden Schleifentypen identisch. Die gleichen Geschwindigkeiten aller Schleifentypen bedeutet für Sie als GFA-BASIC-Programmierer, daß Sie Ihre Schleifentypen ausschließlich nach dem Gesichtspunkt der Programmstrukturierung auswählen können, ohne aus Geschwindigkeitsgründen eine Schleife zu wählen, die für das vorliegende Problem ungeeigneter ist.

4.6 Zeichenketten

In diesem Abschnitt geht es nicht um Geschwindigkeitsoptimierung, sondern nur um eine Verkürzung des entstehenden Programms. Wenn man in einem GFA-BASIC-Programm einer String-Variable eine feste Zeichenkette zuweist, z.B. a$ = "Test", so legt der Compiler diese Zeichenkette ins DATA-Segment des erzeugten Programms.

Der Compiler versucht, den Bereich der initialisierten Strings kurz zu halten. Wenn einer Stringvariablen eine feste Zeichenkette zugewiesen wird, dann durchsucht der Compiler zunächst den bereits bestehenden Zeichenkettenbereich, um festzustellen, ob der neue String bereits in ihm enthalten ist. Wenn dies der Fall ist, so merkt er sich die Adresse des Strings.

Beispiel: Angenommen, Sie möchten ein Programm in mehrere Sprachen übersetzen und packen daher alle Textausgaben in eine Prozedur des BASIC-Programms, um die Übersetzung zu vereinfachen. Dort finden sich jetzt die Zeilen

```
a$="Escape-Taste"
b$="Programm beenden"
x$="Mit der Escape-Taste können Sie das Programm beenden."
```

Im DATA-Segment des Programms wird beim symbolischen Linken das Symbol DATASTAR eingefügt, ab dem Strings abgelegt werden. Dort liegen unsere drei Zeichenketten unmittelbar nacheinander im Speicher.

Sie können die Reihenfolge der drei Zeilen aber abändern in

```
x$="Mit der Escape-Taste können Sie das Programm beenden."
a$="Escape-Taste"
b$="Programm beenden"
```

Nun finden Sie ab DATASTAR nur noch die Zeichenkette "Mit der Escape-Taste können Sie das Programm beenden.". Dies hat folgenden Grund: Zuerst legte der Compiler die lange Textzeile in das DATA-Segment. Anschließend sollte er einen String mit dem Text "Escape-Taste" initialisieren.

Zu diesem Zweck durchsuchte er das DATA-Segment und fand als Bestandteil der langen Textzeile den Text "Escape-Taste", von dem er sich nur die Position im DATA-Segment merkte, ohne ihn neu anzulegen. Das Gleiche geschah mit dem Text "Programm beenden".

Wenn Sie die lange Zeile, wie in der ersten Version als letzte der drei Zeilen aufführen, findet der Compiler im DATA-Segment zum Zeitpunkt der Initialisierung der langen Zeile nur Teile dieser Zeile vor. In einem solchen Fall legt er den kompletten String neu an.

Wenn eine Zeichenkette vorliegt, deren Text in einer anderen Zeichenkette enthalten ist, dann müssen Sie die längere Zeichenkette zuerst einer Variablen zuweisen, damit der Compiler eine Platzoptimierung vornehmen kann.

Strings in DATA-Zeilen werden übrigens noch vor allen Strings, die einer Variablen zugewiesen werden, ab DATASTAR abgelegt. Das DATA-

Segment selbst beginnt mit einer Tabelle für lokale Variablen (VARTAB). Danach folgt die 48 Bytes lange TYPETAB, sofern TYPE oder eine Zuweisung über Pointer (z.B. *x=3 oder SWAP *x,x%()) verwendet wurde.

Hinter der VARTAB oder gegebenenfalls der TYPETAB liegen die INLINE-Bereiche. Erst danach folgt dann das Symbol DATASTAR, ab dem der Inhalt von DATA-Zeilen und initialisierten Strings zu finden ist.

4.7 Lokale und globale Variablen

Im 3.0-Interpreter werden Operationen mit lokalen und globalen Variablen nahezu gleich schnell durchgeführt. Im kompilierten Programm ist das anders. Eine FOR-NEXT-Leerschleife mit 50000 Durchläufen und Vier-Byte-Laufvariable benötigt im Interpreter etwa 1.6 Sekunden. Dies ist unabhängig davon, ob die Laufvariable eine lokale oder globale Variable ist.

Im kompilierten Zustand benötigt die Schleife mit der lokalen Variable etwa 0.7 Sekunden, während die Schleife mit der globalen Variable nur etwa 0.4 Sekunden braucht.

In manchen anderen Sprachen ist dieses Zeitverhältnis bei lokalen und globalen Variablen anders. In C sind z.B. Routinen mit lokalen Variablen in der Regel schneller als die gleichen Routinen mit globalen Variablen.

Notizen

Notizen